Raspberry Pi 5

Dlaczego to nie jest po prostu szybsze Arduino?

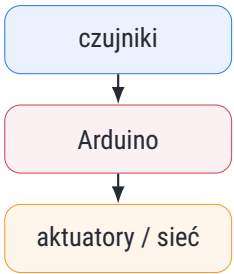
SBC, procesy, GPIO i ograniczenia czasowe



Podstawy Systemów Wbudowanych • Wykład 4
dr inż. Angelika Tefelska

Po trzech wykładach umiemy już zbudować system na mikrokontrolerze

- odczytywać wejścia cyfrowe i analogowe,
- korzystać z przerwań i PWM,
- komunikować się przez UART, I^2C, SPI i 1-Wire,
- obsługiwać czujniki i aktuatory,
- połączyć Arduino z siecią i usługą IoT.



Teraz zmieniamy nie tylko płytkę — zmieniamy **model wykonywania programu**.

To samo zadanie, ale dwa zupełnie różne środowiska

Arduino UNO R4 WiFi

- po resecie startuje nasz firmware,
- program kontroluje mikrokontroler prawie bezpośrednio,
- pamięć i zasoby są ograniczone,
- timing jest względnie przewidywalny.

Raspberry Pi 5

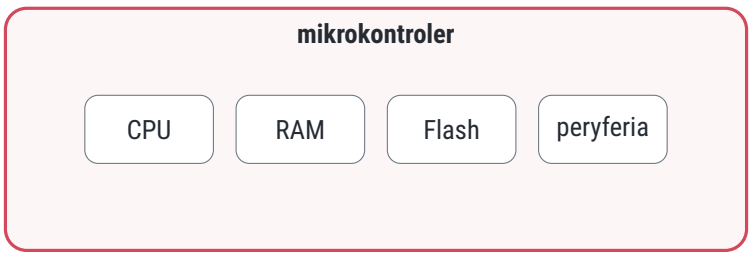
- najpierw startuje bootloader i system operacyjny,
- nasz program jest jednym z wielu procesów,
- mamy dużo pamięci i usług systemowych,
- scheduler może w dowolnym momencie przełączyć CPU na inne zadanie.

1. Mikrokontroler a komputer jednopłytkowy

Najważniejsza różnica nie brzmi „który jest szybszy?”, tylko „kto naprawdę kontroluje sprzęt i czas?”.

MCU — komputer zamknięty w jednym układzie

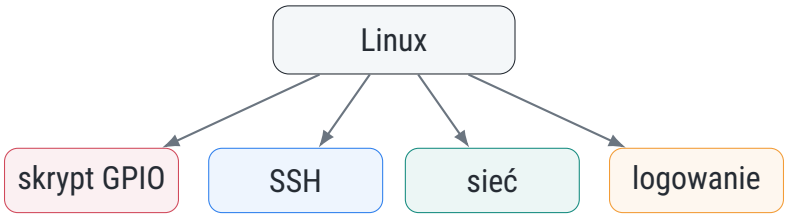
- mikrokontroler integruje rdzeń CPU, pamięć i peryferia w jednym układzie,
- zwykle wykonuje jeden firmware przygotowany dla konkretnego urządzenia,
- GPIO, timery, ADC czy UART są bezpośrednio częścią systemu mikrokontrolera,
- typowym celem jest sterowanie sprzętem przy małym poborze energii i przewidywalnym zachowaniu.



SBC — cały komputer na jednej płytce

- **SBC** = Single-Board Computer,
- ma procesor aplikacyjny, RAM, pamięć masową i rozbudowane interfejsy,
- uruchamia pełny system operacyjny,
- może równocześnie wykonywać wiele programów i usług; system przydziela im czas CPU,
- sprzęt jest udostępniany aplikacjom przez sterowniki i interfejsy systemu operacyjnego.

Raspberry Pi 5 jest bliżej **małego komputera PC** niż Arduino.



MCU vs SBC — porównanie najważniejszych własności

Cecha	Mikrokontroler	Raspberry Pi / SBC
Start	zwykle bardzo szybki	bootloader + system operacyjny
Pamięć	kB–MB	GB RAM + pamięć masowa
Programy	zwykle jeden firmware	wiele procesów jednocześnie
Dostęp do GPIO	bezpośredni przez peryferia MCU	przez kernel / sterowniki / biblioteki
Timing	przewidywalny	zależny od schedulera
System plików	opcjonalny	podstawowy element systemu
Sieć / USB	zależnie od MCU	rozbudowane interfejsy standardowo

Co dzieje się po włączeniu zasilania?

Arduino

reset → inicjalizacja → `setup()` → `loop()`

Raspberry Pi

bootloader → kernel Linux → usługi → login / aplikacje

Na Raspberry Pi nasz skrypt pojawia się **dopiero na końcu** długiego łańcucha.

Pamięć na MCU i Raspberry Pi — RAM, pamięć wirtualna i pliki programu

Mikrokontroler

- rozmiar RAM i Flash jest z góry określony,
- firmware po kompilacji zapisujemy w pamięci programu,
- duże bufory szybko zużywają ograniczony RAM,
- program zwykle pracuje bez systemu pamięci wirtualnej.

Raspberry Pi / Linux

- każdy proces ma własną **wirtualną przestrzeń adresową**,
- kernel mapuje ją na fizyczny RAM i izoluje procesy,
- skrypt `.py` jest plikiem w systemie plików,
- interpreter Python i nasz skrypt są osobnymi elementami uruchamianymi przez system.

Na Raspberry Pi kod, interpreter, RAM i pamięć masowa są zarządzane przez **system operacyjny**.

Zużycie energii i zasilanie też są częścią wyboru platformy

- MCU może pracować z baterii miesiącami lub latami dzięki trybom uśpienia,
- Raspberry Pi 5 jest pełnym komputerem i potrzebuje znacznie większego budżetu energetycznego,
- dla Raspberry Pi 5 producent zaleca zasilanie USB-C 5 V / 5 A; zasilanie 5 V / 3 A może ograniczać budżet dla peryferiów USB,
- przy projektowaniu urządzenia „wydajniejszy” nie zawsze znaczy „lepszy”.

Źródło: Raspberry Pi Documentation — Getting started

Kiedy mikrokontroler, a kiedy Raspberry Pi?

Mikrokontroler

- precyzyjne generowanie i pomiar impulsów,
- szybki start i przewidywalny czas reakcji,
- niski pobór energii,
- prosta, powtarzalna funkcja urządzenia,
- niewielki koszt i mała liczba komponentów.

Przykłady: regulator silnika, sterownik LED, węzeł bateryjny.

Raspberry Pi

- system plików, baza danych, duże biblioteki,
- kamera, obraz, audio i większe strumienie danych,
- USB, Ethernet, interfejs użytkownika i usługi sieciowe,
- kilka programów pracujących równocześnie,
- brak wymogu reakcji z gwarancją rzędu mikrosekund.

Przykłady: gateway IoT, serwer, rejestrator danych, analiza obrazu.

Często najlepsza jest architektura hybrydowa



- MCU realizuje krytyczne czasowo sterowanie,
- Raspberry Pi wykonuje cięższe obliczenia i komunikację,
- każda platforma robi to, do czego została zaprojektowana.

Które zadanie najbardziej naturalnie pasuje do mikrokontrolera?

1. analiza obrazu 4K i zapis dużych plików
2. uruchomienie serwera WWW i bazy danych
3. precyzyjne generowanie impulsu co 10 μs
4. praca z monitorem HDMI i klawiaturą

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Które zadanie najbardziej naturalnie pasuje do mikrokontrolera?

1. analiza obrazu 4K i zapis dużych plików
2. uruchomienie serwera WWW i bazy danych
3. **precyzyjne generowanie impulsu co 10 μs**
4. praca z monitorem HDMI i klawiaturą

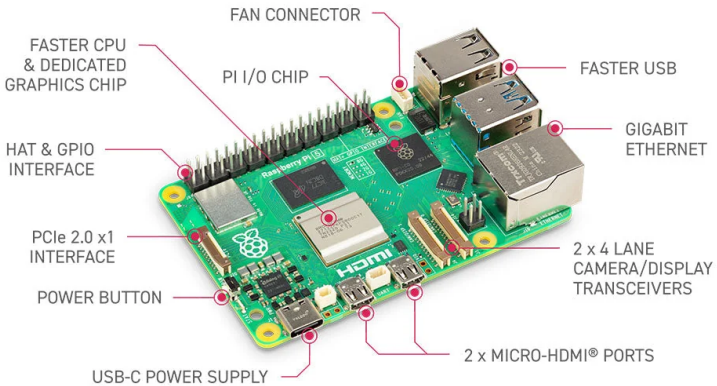
?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

2. Raspberry Pi 5 — co właściwie znajduje się na płytce?

Poznajemy elementy sprzętowe, ale tylko te, które później wpływają na program i laboratorium.

Raspberry Pi 5 — najważniejsze elementy



Źródło: https://kitronik.co.uk/

- 64-bitowy, czterordzeniowy Arm Cortex-A76 @ 2.4 GHz,
- ma pamięci cache L2 i współdzieloną L3,
- współpracuje z zewnętrzną pamięcią RAM typu LPDDR4X,
- został zaprojektowany do uruchamiania złożonego oprogramowania i systemu operacyjnego.

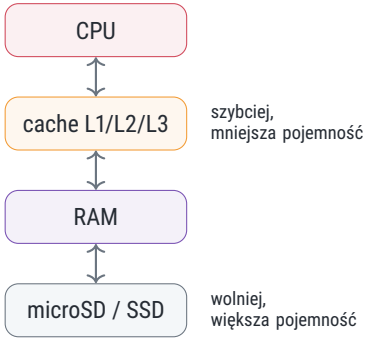
Wiele rdzeni i pamięć cache — skąd bierze się wydajność?

Cztery rdzenie CPU

- procesów i wątków może być znacznie więcej niż rdzeni,
- scheduler przydziela im czas na dostępnych rdzeniach,
- wątek może zostać przeniesiony pomiędzy rdzeniami,
- wielordzeniowość zwiększa przepustowość, ale nie gwarantuje czasu rzeczywistego.

Po co cache?

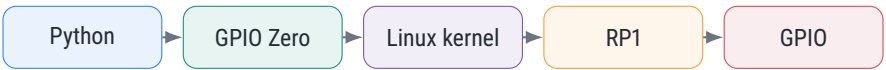
CPU szuka często używanych danych najpierw w małej, szybkiej pamięci cache. Dopiero gdy ich tam nie ma, sięga do wolniejszego RAM-u.



Różne czasy dostępu do danych są jedną z przyczyn zmiennego czasu wykonania programu.

RP1 — układ I/O charakterystyczny dla Raspberry Pi 5

- **BCM2712** to główny procesor aplikacyjny Raspberry Pi 5 — zawiera m.in. cztery rdzenie Arm Cortex-A76 i kontroler pamięci,
- **RP1** to osobny kontroler wejścia/wyjścia zaprojektowany przez Raspberry Pi,
- RP1 łączy się z BCM2712 szybkim łączem PCIe i obsługuje dużą część zewnętrznych interfejsów, m.in. GPIO i USB,
- dlatego droga od programu do GPIO w Raspberry Pi 5 różni się od wcześniejszych generacji i od mikrokontrolera.



RAM — dużo większy, ale współdzielony przez cały system

- Raspberry Pi 5 korzysta z pamięci **LPDDR4X** — jest to rodzaj szybkiej, energooszczędnej pamięci DRAM używanej jako pamięć operacyjna; dla nas najważniejsze jest po prostu: **to RAM komputera**,
- RAM zajmuje kernel, cache dyskowy, usługi, środowisko graficzne i nasze procesy,
- pamięć jednego procesu jest chroniona przed przypadkowym zapisem przez inny proces,
- gdy zaczyna brakować RAM-u, system może korzystać z pamięci masowej, co jest znacznie wolniejsze.

RAM

- szybki i ulotny,
- przechowuje aktualnie używany kod i dane.

microSD / SSD

- wolniejsze, ale nieulotne,
- przechowują system, programy i pliki.

Szybkie interfejsy i łączność Raspberry Pi 5

PCIe

- szybkie łącze do zewnętrznych urządzeń,
- Raspberry Pi 5 udostępnia linię PCIe dla akcesoriów,
- z adapterem można podłączyć np. dysk NVMe.

USB i Ethernet

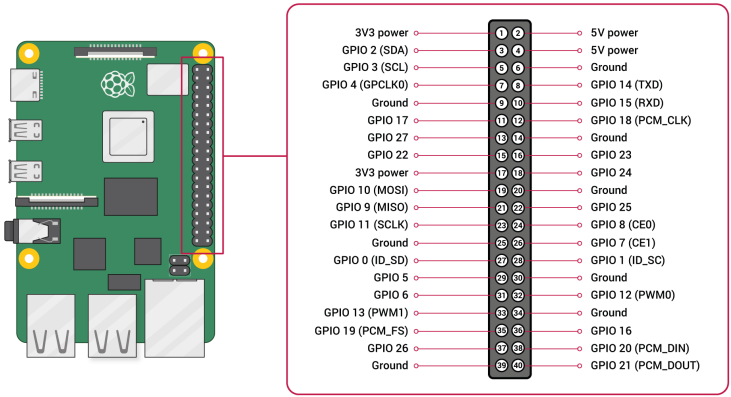
- 2× USB 3.0 i 2× USB 2.0,
- Gigabit Ethernet,
- sterowniki Linuxa obsługują wiele gotowych urządzeń USB.

Łączność radiowa

- dwupasmowe Wi-Fi 802.11ac,
- Bluetooth / BLE,
- jedna konfiguracja sieci jest współdzielona przez wiele procesów.

SBC korzysta z gotowego **ekosystemu interfejsów i sterowników**, zamiast implementować każdy stos samodzielnie.

40-pinowe złącze GPIO



Źródło: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

Wydajność oznacza też ciepło

- podczas większego obciążenia CPU i I/O rośnie pobór mocy,
- system monitoruje temperaturę i może ograniczać częstotliwość, gdy jest zbyt gorąco,
- w zastosowaniach obciążających Pi warto przewidzieć chłodzenie i przepływ powietrza,
- termika może wpływać na wydajność długotrwałych obliczeń.



Źródło: https://botland.com.pl/

Który układ w Raspberry Pi 5 obsługuje dużą część zewnętrznego I/O?

1. pamięć LPDDR4X
2. karta microSD
3. wyłącznie rdzeń Cortex-A76
4. RP1

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Który układ w Raspberry Pi 5 obsługuje dużą część zewnętrznego I/O?

1. pamięć LPDDR4X
2. karta microSD
3. wyłącznie rdzeń Cortex-A76
4. **RP1**

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

3. Linux — warstwa pomiędzy programem a sprzętem

System operacyjny daje nam procesy, pliki, sieć i sterowniki, ale odbiera aplikacji pełną kontrolę nad czasem.

Po co system operacyjny i czym jest Raspberry Pi OS?

Rola systemu operacyjnego

- zarządza CPU, pamięcią i urządzeniami,
- uruchamia wiele programów równocześnie,
- zapewnia system plików, sieć, użytkowników i uprawnienia,
- udostępnia aplikacjom API i sterowniki zamiast bezpośredniego dostępu do sprzętu.

Raspberry Pi OS

- oficjalny system dla komputerów Raspberry Pi,
- oparty na Debianie,
- może działać z pulpitem lub **headless**.

Co znaczy headless?

Komputer pracuje **bez monitora, klawiatury i lokalnego pulpitu**. Konfigurujemy go i uruchamiamy programy zdalnie, najczęściej przez SSH.

Boot Raspberry Pi 5 — uproszczony łańcuch



- Raspberry Pi 5 ma bootloader w pamięci EEPROM na płytce,
- bootloader wybiera nośnik i ładuje system,
- dopiero potem uruchamiane są procesy użytkownika.

Kernel i user space — dwie strefy

Kernel space

- sterowniki,
- scheduler,
- pamięć,
- system plików,
- sieć.

User space

- shell,
- Python,
- edytor,
- nasze skrypty,
- usługi użytkownika.

Proces — uruchomiona instancja programu

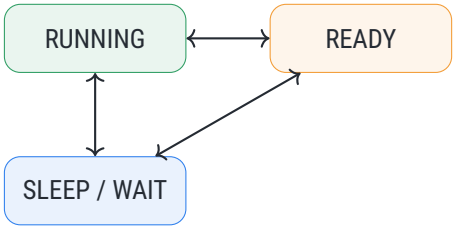
- plik programu na dysku nie jest jeszcze procesem,
- po uruchomieniu proces dostaje własną pamięć, identyfikator PID i zasoby,
- ten sam skrypt możemy uruchomić kilka razy — powstaną różne procesy,
- system śledzi ich stan i czas CPU.

Przykład

`python3 blink.py` może mieć PID 2147, a drugi proces `python3 logger.py` PID 2153.

- PID wykorzystujemy do diagnostyki i kończenia procesu,
- nazwa programu nie zawsze wystarcza — może istnieć wiele instancji.

Stan procesu — program nie cały czas wykonuje instrukcje



- proces może czekać na CPU, dane, timer albo urządzenie,
- „program działa” nie oznacza, że w każdej chwili ma rdzeń CPU.

Scheduler — kto dostaje CPU?

- scheduler kernela wybiera, który wątek ma aktualnie wykonywać instrukcje,
- uwzględnia priorytety, stan procesów i dostępne rdzenie,
- może przerwać jeden proces i uruchomić inny,
- zwykła aplikacja nie kontroluje dokładnie chwili, w której zostanie ponownie uruchomiona.

Proces a wątek — dwie różne rzeczy

- proces ma własną przestrzeń pamięci i zasoby,
- jeden proces może mieć wiele wątków wykonania,
- wątki tego samego procesu współdzielą pamięć,
- biblioteki mogą tworzyć wątki w tle, nawet jeśli nasz kod wygląda liniowo.

Przykład później

GPIO Zero potrafi wykonywać niektóre działania w tle — dlatego np. `led.blink()` może zwrócić sterowanie do programu.

W Linuxie „wszystko jest plikiem” — ważna intuicja

- zwykłe pliki to tylko jeden przypadek,
- urządzenia i interfejsy kernela często są reprezentowane w hierarchii plików,
- przykładowe katalogi: `/dev`, `/proc`, `/sys`,
- odczyt/zapis pliku może w rzeczywistości uruchamiać operację sterownika.

Przykłady

`/dev/ttyUSB0` — interfejs szeregowy przez USB
`/dev/gpiochip0` — kontroler GPIO udostępniony przez kernel

- nazwa w `/dev` jest interfejsem do sterownika, a nie „plikiem danych” na microSD,
- aplikacja korzysta z niego zgodnie z API kernela.

Uprawnienia chronią system i urządzenia

- Linux rozróżnia użytkowników i grupy,
- plik lub urządzenie ma właściciela i zestaw uprawnień,
- brak dostępu do urządzenia może być problemem **uprawnień**, a nie błędem kodu,
- uruchamianie wszystkiego przez `sudo` nie jest dobrym sposobem diagnozowania.

systemd — kto uruchamia usługi po starcie?

systemd

- po uruchomieniu kernela startuje system użytkowy i usługi,
- w Raspberry Pi OS rolę menedżera usług pełni systemd,
- może automatycznie uruchamiać sieć, SSH i nasze własne usługi,
- usługa może być restartowana i monitorowana niezależnie od terminala użytkownika.

Logi

- w systemie headless nie zawsze mamy otwarty terminal z `print()`,
- system i usługi zapisują zdarzenia w logach,
- `journalctl` pozwala sprawdzić komunikaty usług zarządzanych przez systemd,
- logi są podstawowym narzędziem diagnostycznym urządzenia pracującego samodzielnie.

Przypomnienie podstaw Linux-a

```
pwd
/home/student
```

- każdy proces ma bieżący katalog roboczy,
- ścieżki względne są interpretowane względem niego,
- wiele błędów „file not found” wynika z uruchomienia programu w innym katalogu.

```
ls
ls -l
ls -la
```

- `-l`: szczegóły, właściciel, uprawnienia, rozmiar,
- `-a`: także pliki ukryte rozpoczynające się od kropki.

Przypomnienie podstaw Linux-a

```
cd /home/student/psw
cd ..
cd ~
```

- ~ oznacza katalog domowy użytkownika,
- po zmianie katalogu warto sprawdzić pwd.

```
mkdir lab8
cp blink.py lab8/
mv test.py lab8/button.py
```

- `mkdir` — katalog,
- `cp` — kopia,
- `mv` — przeniesienie lub zmiana nazwy.

```
rm test.txt
rm -r stary_katalog/
```

Uwaga

W terminalu nie ma typowego „kosza”. Polecenie wykonane z błędną ścieżką może bezpowrotnie usunąć dane.

```
cat config.txt
less duzy_log.txt
head -n 20 duzy_log.txt
tail -n 20 duzy_log.txt
tail -f log.txt
```

- `head -n 20` pokazuje **pierwsze 20 linii** pliku,
- `tail -n 20` pokazuje **ostatnie 20 linii**,
- `tail -f` pozostaje uruchomione i pokazuje nowe linie dopisywane do logu.

Przypomnienie podstaw Linux-a

```
grep "error" log.txt
grep -i "gpio" log.txt
```

- `grep` filtruje linie pasujące do wzorca,
- `-i` ignoruje wielkość liter.

```
python3 sensor.py > wynik.txt
python3 sensor.py >> historia.txt
```

- `>` nadpisuje plik,
- `»` dopisuje na końcu,
- program nie musi wiedzieć, że jego standardowe wyjście trafiło do pliku.

Przypomnienie podstaw Linux-a

```
-rwxr-xr-- 1 student student 1200 blink.py
```

- `r` — read, `w` — write, `x` — execute,
- trzy zestawy dotyczą kolejno: **właściciela, grupy, pozostałych**.

```
chmod u+x skrypt.sh
chmod 644 dane.txt
```

- zapis symboliczny: `u+x` = dodaj właścicielowi prawo wykonania,
- w zapisie liczbowym: `r=4`, `w=2`, `x=1`,
- `644`: właściciel `6=4+2=rw`, grupa `4=r`, pozostali `4=r`.

Przypomnienie podstaw Linux-a

```
ps aux
top
```

- `ps` pokazuje migawkę procesów,
- w popularnym zapisie `aux`: `a` rozszerza listę na procesy innych użytkowników z terminalem, `x` dodaje także procesy bez terminala, a `u` włącza szczegółowy format z użytkownikiem i użyciem zasobów,
- `top` odświeża listę dynamicznie.

```
kill 2147
kill -9 2147
```

- zwykłe `kill` wysyła żądanie zakończenia,
- `-9` wymusza zakończenie bez możliwości posprzątania zasobów,
- `-9` zostawiamy na sytuacje, gdy łagodniejsze zakończenie nie działa.

systemctl — zarządzanie usługami

```
systemctl status ssh
sudo systemctl restart ssh
```

- usługa może działać bez otwartego terminala,
- status pokazuje, czy wystartowała poprawnie,
- później ten sam mechanizm może uruchamiać własną aplikację IoT po starcie urządzenia.

journalctl — logi usług systemowych

```
journalctl -u ssh
journalctl -u ssh -n 30
```

- przy awarii usługi zaczynamy od statusu i logów,
- to lepsze niż „restartuj aż zadziała”.

SSH — terminal Raspberry Pi z własnego komputera

```
ssh student@192.168.1.50
```

- SSH daje zdalną powłokę,
- Raspberry Pi OS domyślnie nie musi mieć serwera SSH włączonego — można aktywować go podczas konfiguracji lub później,
- po połączeniu wszystkie polecenia wykonują się na Raspberry Pi.

scp — kopiowanie plików przez SSH

```
scp blink.py student@192.168.1.50:/home/student/psw/
```

- plik kopiujemy bezpośrednio przez sieć na działający Raspberry Pi,
- transfer jest szyfrowany,
- przydaje się do przesyłania kodu i wyników.

Kilka komend sieciowych, które warto pamiętać

```
hostname -I
ip addr
ping 192.168.1.1
```

- `hostname -I` — szybki podgląd adresów IP,
- `ip addr` — pełniejsze informacje o interfejsach,
- `ping` — prosty test osiągalności hosta.

apt — oprogramowanie jako pakiety systemowe

```
sudo apt update
sudo apt install python3-gpiozero
```

- repozytorium systemu zawiera kompatybilne pakiety,
- aktualizację listy pakietów wykonujemy przed instalacją,
- dla bibliotek związanych ze sprzętem często warto preferować pakiety systemowe Raspberry Pi OS.

Dlaczego zwykły skrypt Python na Raspberry Pi nie ma pełnej kontroli nad czasem CPU?

1. bo scheduler Linuxa współdzieli CPU pomiędzy wiele zadań
2. bo Raspberry Pi ma tylko jeden rdzeń
3. bo Python nie potrafi obsługiwać GPIO
4. bo microSD zawsze blokuje procesor

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Dlaczego zwykły skrypt Python na Raspberry Pi nie ma pełnej kontroli nad czasem CPU?

1. **bo scheduler Linuxa współdzieli CPU pomiędzy wiele zadań**
2. bo Raspberry Pi ma tylko jeden rdzeń
3. bo Python nie potrafi obsługiwać GPIO
4. bo microSD zawsze blokuje procesor

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Które polecenie najlepiej sprawdzi, czy proces Python nadal działa?

1. pwd
2. ps aux | grep python
3. chmod 644
4. mkdir python

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Które polecenie najlepiej sprawdzi, czy proces Python nadal działa?

1. pwd
2. **ps aux | grep python**
3. chmod 644
4. mkdir python

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

5. GPIO w Raspberry Pi 5 - GPIO Zero w praktyce

Ten sam pin HIGH/LOW, ale zupełnie inna droga programowa niż w mikrokontrolerze.

Najpierw numeracja: fizyczny pin to nie numer GPIO

Numer fizyczny

Pozycja na 40-pinowym złączu, np. **pin 11**. Liczymy kolejne styki na złączu.

Numer BCM / GPIO

Logiczna nazwa linii GPIO używana w kodzie, np. **GPIO17**. Nazwa „BCM” pochodzi z historycznej konwencji numeracji Raspberry Pi i jest nadal powszechnie używana przez biblioteki.

Pin fizyczny	BCM	Przykładowa rola
1	—	3.3 V
2	—	5 V
6	—	GND
11	GPIO17	zwykłe GPIO
13	GPIO27	zwykłe GPIO
15	GPIO22	zwykłe GPIO
3	GPIO2	I^2C SDA
5	GPIO3	I^2C SCL

Na Raspberry Pi 5 fizyczne GPIO obsługuje RP1, ale dla programisty nadal używamy nazw takich jak GPIO17.

Pin fizyczny 11 $\neq$ GPIO11. Pin fizyczny 11 odpowiada GPIO17.

pinout — szybka mapa złącza na konkretnym Raspberry Pi

```
pinout
```

- pokazuje 40-pinowe złącze i funkcje pinów,
- ogranicza ryzyko pomyłki pomiędzy numerem fizycznym i BCM,
- jest wygodne przy pracy z terminala bez otwierania dokumentacji w przeglądarce.

GPIO jako urządzenie znakowe Linuxa

- aplikacja działa w user space i nie powinna omijać kernela,
- architektura Raspberry Pi 5 z RP1 różni się od wcześniejszych modeli,
- bezpośrednie adresy rejestrów są zależne od konkretnego hardware'u,
- nowoczesne API GPIO w Linuxie korzysta z urządzeń `gpiochip`.

Typowy widok

```
/dev/gpiochip0
```

- kernel udostępnia kontroler GPIO przez ustandaryzowany interfejs,
- biblioteka użytkownika może rezerwować linie, ustawiać kierunek i czekać na zdarzenia,
- dzięki temu aplikacja nie musi znać fizycznych rejestrów RP1.

libgpiod — narzędzia do GPIO w Linuxie

- `gpiodetect` — lista kontrolerów GPIO,
- `gpioinfo` — informacje o liniach i ich stanie,
- `gpioget` — odczyt wartości linii,
- `gpioset` — ustawienie wyjścia,
- `gpiomon` — oczekiwanie na zbocza.

Po co je znać?

Nie musimy nimi pisać całej aplikacji. Są bardzo przydatne do **diagnostyki**: czy kernel widzi GPIO, czy linia jest zajęta i czy stan wejścia rzeczywiście się zmienia.

```
$ gpiodetect
gpiochip0 [pinctrl-rp1] (... lines)

$ gpioinfo GPIO17
... GPIO17 ... input ...

$ gpioget --numeric GPIO17
0

$ gpiomon GPIO17
... rising ...
... falling ...
```

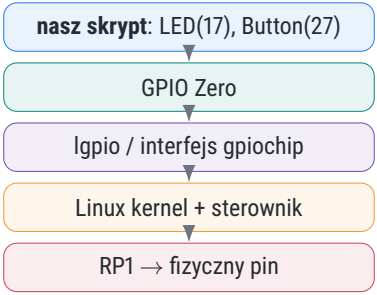
Wynik jest przykładowy — numer `gpiochip` i szczegóły formatu mogą zależeć od wersji kernela i libgpiod.

GPIO Zero — warstwa wysokiego poziomu nad GPIO

- tworzymy obiekty odpowiadające elementom: `LED`, `Button`, `PWMLED`, `Motor` itd.,
- kod opisuje **intencję**: „włącz LED”, „czekaj na przycisk”, a nie „ustaw bit rejestru”,
- **pin factory** to warstwa łącząca GPIO Zero z niższym API GPIO,
- dla Raspberry Pi 5 aktualna dokumentacja wskazuje backend `lgpio`.

Korzyść

Kod wysokiego poziomu nie musi znać mapy rejestrów RP1 ani szczegółów sterownika kernela.



GPIO Zero — jakie elementy potrafi reprezentować?

Grupa	Przykładowe klasy	Do czego służą
Wyjścia cyfrowe	`LED`, `Buzzer`, `DigitalOutputDevice`	włącz/wyłącz, toggle, miganie
Wyjścia z PWM	`PWMLED`, `PWMOutputDevice`	jasność / wartość 0...1, płynne przejścia
Wejścia	`Button`, `MotionSensor`, `LineSensor`	stan, zdarzenia, oczekiwanie na aktywację
Czujniki złożone	`DistanceSensor`	pomiar HC-SR04, próg odległości, zdarzenia
Aktuatory złożone	`Motor`, `Servo`, `AngularServo`	kierunek i prędkość silnika, położenie serwa
Grupy elementów	`LEDBoard`, `RGBLED`, `Robot`	kilka GPIO reprezentowanych jednym obiektem

Jedna biblioteka daje wspólny styl pracy z bardzo różnymi elementami.

Źródło: GPIO Zero Documentation

GPIO Zero - jak zapalić diodę LED?

```
1  from gpiozero import LED
2
3  led = LED(17)
4  led.on()
5
```

- 17 oznacza GPIO17 w numeracji BCM,
- obiekt `led` reprezentuje fizyczny element podłączony do pinu,
- `on()` ustawia aktywny stan wyjścia.

Dlaczego LED może zgasnąć po zakończeniu programu?

- po zakończeniu procesu biblioteka zwalnia zasoby,
- stan pinu może zostać zresetowany,
- dlatego program sterujący sprzętem zwykle musi pozostać aktywny,
- „ustawiłam pin raz” nie oznacza, że proces może zawsze natychmiast się zakończyć.

Migająca dioda LED — wersja jawna

```python
from gpiozero import LED
from time import sleep

led = LED(17)

while True:
    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

Migająca dioda LED — GPIO Zero potrafi uruchomić działanie w tle

```
from gpiozero import LED
from signal import pause

led = LED(17)
led.blink()
pause()
```

- `blink()` domyślnie działa w tle,
- `pause()` utrzymuje główny proces przy życiu,
- to już przykład współbieżności ukrytej przez bibliotekę.

Polling — najprostszy sposób odczytu przycisku

```python
from gpiozero import Button
from time import sleep

button = Button(27)

while True:
    if button.is_pressed:
        print("Wcisniety")
    sleep(0.05)
```

- przycisk jest podłączony przez rezystory *pull-up*,
- program regularnie sprawdza stan,
- łatwe do zrozumienia, ale CPU wykonuje pętlę nawet wtedy, gdy nic się nie dzieje.

Zdarzenie — wykonaj funkcję wtedy, gdy przycisk zostanie naciśnięty

```python
from gpiozero import Button
from signal import pause

def pressed():
    print("Przycisk!")

button = Button(27)
button.when_pressed = pressed

pause()
```

Poprawnie

`button.when_pressed = pressed`

Przekazujemy funkcję, którą biblioteka wywoła później.

Nie to

`button.when_pressed = pressed()`

Wywołujemy funkcję natychmiast i przypisujemy jej wynik.

Przycisk steruje LED — callbacki

```python
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(27)

button.when_pressed = led.on
button.when_released = led.off

pause()
```

- biblioteka reaguje na zdarzenia wejścia,
- nie potrzebujemy własnej nieskończonej pętli odczytu.

GPIO Zero pozwala też połączyć źródło i odbiornik deklaratywnie

```python
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(27)

led.source = button
pause()
```

- deklarujemy zależność zamiast pisać pętlę,
- biblioteka aktualizuje wartość w tle,
- dobry przykład, jak biblioteka wysokiego poziomu zmienia styl programowania sprzętu.

Button z programowym debounce

- system operacyjny nie usuwa fizycznych drgań styków,
- jedno naciśnięcie może wygenerować kilka zboczy,
- GPIO Zero pozwala ustawić parametr `bounce_time`,
- to wygodna warstwa programowej filtracji zdarzeń.

```
1 from gpiozero import Button
2 from signal import pause
3
4 button = Button(27, bounce_time=0.05)
5
6 button.when_pressed = lambda: print("PRESS")
7 pause()
```

50 ms jest wartością przykładową — dobieramy ją do rzeczywistego elementu i zastosowania.

PWMLED — jasność jako wartość 0...1

```python
from gpiozero import PWMLED
from time import sleep

led = PWMLED(17)

led.value = 0.2
sleep(1)
led.value = 0.7
sleep(1)
led.value = 1.0
```

- API operuje na wartości znormalizowanej 0...1,
- biblioteka generuje przebieg PWM przez odpowiednią warstwę backendu.

PWM na Linuxie — ważne ograniczenie

Nie zakładamy precyzji mikrokontrolera

Jeżeli PWM jest generowany przez warstwy programowe, momenty przełączeń mogą czasem wypaść nieco wcześniej lub później, ponieważ system operacyjny może opóźnić wykonanie programu.

- do LED lub prostego sterowania często jest to wystarczające,
- do krytycznego czasowo sterowania silnikiem lub protokołu bitowego lepiej użyć sprzętowego peryferium albo MCU.

Obsługa błędów sprzętowych — program powinien kończyć się czytelnie

```
from gpiozero import LED

try:
  led = LED(17)
  led.on()
  input("Enter, aby zakonczyc...")
finally:
  led.close()

```

- zasób powinien zostać zwolniony również przy błędzie,
- w większych aplikacjach porządek przy zamykaniu procesu jest ważny.

LED i PWMLED — więcej niż on() i off()

```python
from gpiozero import LED
from signal import pause

led = LED(17)
led.on()
led.toggle()
led.blink(on_time=0.2, off_time=0.8, n=5)
pause()
```

```python
from gpiozero import PWMLED
from signal import pause

led = PWMLED(17)
led.value = 0.35
led.pulse(fade_in_time=0.5, fade_out_time=0.5)
pause()
```

- `blink()` przyjmuje m.in. czas świecenia, zgaszenia i liczbę powtórzeń,
- `PWMLED.value` ma zakres 0...1,
- `pulse()` realizuje płynne rozjaśnianie i wygaszanie w tle.

Button — pojedyncze kliknięcie, puszczenie i przytrzymanie

```python
from gpiozero import Button
from signal import pause

button = Button(27, bounce_time=0.05, hold_time=1.5)

button.when_pressed = pressed
button.when_released = released
button.when_held = long_press

pause()
```

- `is_pressed` — aktualny stan,
- `when_pressed` — zdarzenie naciśnięcia,
- `when_released` — zdarzenie puszczenia,
- `when_held` — reakcja po przytrzymaniu,
- `hold_time` — po jakim czasie uznajemy przycisk za przytrzymany,
- `wait_for_press()` — blokujące oczekiwanie na naciśnięcie.

Ultradźwiękowy czujnik odległości - HC-SR04

```python
from gpiozero import DistanceSensor, LED
from signal import pause

sensor = DistanceSensor(
    echo=18,
    trigger=23,
    threshold_distance=0.20)
led = LED(17)

sensor.when_in_range = led.on
sensor.when_out_of_range = led.off

print(sensor.distance)  # metry
pause()
```

- nie mierzymy ręcznie czasu impulsu ECHO,
- `distance` zwraca odległość w metrach,
- próg 0.20 m generuje zdarzenia `in_range`/`out_of_range`,
- kod opisuje zachowanie systemu, a szczegóły pomiaru ukrywa biblioteka.

HC-SR04 + Raspberry Pi

ECHO czujnika ma poziom 5 V. Wejście GPIO Raspberry Pi jest 3.3 V, dlatego potrzebny jest dzielnik napięcia / konwerter poziomów.

Motor i AngularServo — złożony element jako jeden obiekt

```python
from gpiozero import Motor
from time import sleep

motor = Motor(forward=17,
              backward=18)

motor.forward(0.5)
sleep(2)
motor.backward(0.3)
sleep(2)
motor.stop()
```

`forward()` i `backward()` przyjmują prędkość 0...1, gdy używamy PWM.

```python
from gpiozero import AngularServo
from time import sleep

servo = AngularServo(
    12,
    min_angle=-90,
    max_angle=90)

servo.angle = -45
sleep(1)
servo.angle = 45
```

GPIO dostarcza **sygnał sterujący**. Silnika ani większego serwa nie zasilamy bezpośrednio z pinu GPIO.

Co oznacza LED(17) w GPIO Zero?

1. GPIO17 w numeracji BCM
2. fizyczny pin 17 złącza
3. 17-procentowe wypełnienie PWM
4. 17 V na wyjściu

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Co oznacza LED(17) w GPIO Zero?

1. **GPIO17 w numeracji BCM**
2. fizyczny pin 17 złącza
3. 17-procentowe wypełnienie PWM
4. 17 V na wyjściu

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Po co na Raspberry Pi 5 korzystamy ze wspieranego API GPIO?

1. aby odizolować kod od różnic sprzętowych między generacjami, np. od obecności RP1
2. ponieważ bezpośredni dostęp do rejestrów jest zawsze wolniejszy od biblioteki
3. ponieważ tylko GPIO Zero pozwala używać numeracji BCM
4. ponieważ każde GPIO w Linuxie musi działać jako osobna usługa systemd

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Odpowiedź

Poprawna odpowiedź: 1

Warstwa kernela i wspierane biblioteki oddzielają aplikację od szczegółów konkretnego kontrolera GPIO. Raspberry Pi 5 korzysta z RP1, więc kod zakładający mapę rejestrów wcześniejszej generacji może być po prostu niepoprawny.

- biblioteka nie jest automatycznie „szybsza” od dostępu niskopoziomowego,
- numeracja GPIO nie jest własnością wyłącznie GPIO Zero,
- GPIO może być używane przez zwykły proces — nie wymaga osobnej usługi systemd.

7. Timing i czas rzeczywisty

Raspberry Pi jest bardzo szybkie, ale szybkość i deterministyczność to dwie różne własności.

Szybki system może nie być deterministyczny

Przykład

Operacja zwykle kończy się po 50 µs, ale raz na jakiś czas po 2 ms, bo proces został wywłaszczony przez system.

- średnia może wyglądać świetnie,
- ale dla deadline'u 100 µs system zawodzi.
- wynik spóźniony może być równie bezużyteczny jak wynik błędny,
- istotny jest najgorszy przypadek, nie tylko średni czas wykonania.

Latency i jitter — dwa pojęcia, które warto rozróżnić

Latency

Opóźnienie od zdarzenia do reakcji.

Jitter

Zmiana tego opóźnienia pomiędzy kolejnymi zdarzeniami.

W systemie czasu rzeczywistego często walczymy bardziej z **najgorszym jitterem** niż ze średnią szybkością.

sleep(0.001) nie oznacza „wykonaj dokładnie po 1 ms”

- `sleep()` mówi systemowi: nie planuj mnie do wykonania przed upływem tego czasu,
- po upływie czasu proces staje się gotowy,
- scheduler może uruchomić go nieco później,
- dokładne opóźnienie zależy od obciążenia i działania kernela.



Co może opóźnić nasz proces?

- inny proces dostał CPU,
- obsługa przerwania sprzętowego,
- operacja systemu plików lub sterownika,
- aktywność sieci,
- migracja wątku między rdzeniami i efekty cache,
- zarządzanie pamięcią.

Czy to znaczy, że Raspberry Pi nie nadaje się do sterowania?

Nie. Trzeba dobrać skalę czasu i wymagania.

Sterowanie LED, odczyt przycisku, pomiary co sekundę, MQTT, UI czy logowanie danych są naturalnymi zadaniami dla Raspberry Pi.

Problem

Precyzyjny protokół bitowy lub pętla sterowania wymagająca gwarantowanej reakcji w mikrosekundach to inna klasa problemu.

Przykłady zadań:

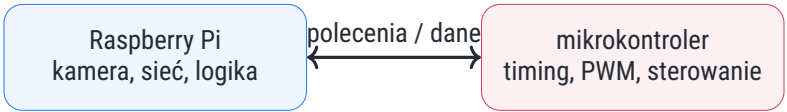
Dobrze pasujących do Raspberry Pi

- pomiar temperatury co 1 s,
- zapis danych do pliku lub bazy,
- dashboard i serwer WWW,
- kamera i analiza obrazu,
- sterowanie oświetleniem z opóźnieniem rzędu dziesiątek ms,
- komunikacja MQTT i integracja kilku urządzeń.

Wymagających ostrożności

- bardzo precyzyjne impulsy czasowe,
- szybka pętla regulacji silnika,
- generacja przebiegów o małym jitterze,
- system bezpieczeństwa z twardym deadline'em reakcji.

Gdy wymagania są krytyczne — delegujemy zadanie do MCU



Raspberry Pi mówi **co zrobić**; MCU pilnuje **kiedy dokładnie**.

Jak połączyć Raspberry Pi z mikrokontrolerem?

- USB / wirtualny port szeregowy,
- UART,
- I^2C,
- SPI,
- sieć — jeżeli oba urządzenia ją mają.

Dobór interfejsu zależy od szybkości, odległości, złożoności i wymaganej niezawodności.

Które zdanie najlepiej opisuje zwykły Raspberry Pi OS?

1. gwarantuje dokładny czas reakcji każdego procesu
2. jest systemem hard real-time
3. zapewnia wysoką wydajność, ale zwykły proces może mieć zmienny czas reakcji
4. nie potrafi wykonywać kilku procesów równocześnie

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Które zdanie najlepiej opisuje zwykły Raspberry Pi OS?

1. gwarantuje dokładny czas reakcji każdego procesu
2. jest systemem hard real-time
3. **zapewnia wysoką wydajność, ale zwykły proces może mieć zmienny czas reakcji**
4. nie potrafi wykonywać kilku procesów równocześnie

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Co jest najważniejszą różnicą przy przejściu z Arduino na Raspberry Pi?

1. Raspberry Pi nie ma GPIO
2. na Raspberry Pi nie można korzystać z czujników
3. Raspberry Pi ma więcej RAM, więc program nie podlega już żadnym ograniczeniom zasobów
4. na Raspberry Pi nasz program działa jako proces pod kontrolą systemu operacyjnego

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Co jest najważniejszą różnicą przy przejściu z Arduino na Raspberry Pi?

1. Raspberry Pi nie ma GPIO
2. na Raspberry Pi nie można korzystać z czujników
3. Raspberry Pi ma więcej RAM, więc program nie podlega już żadnym ograniczeniom zasobów
4. **na Raspberry Pi nasz program działa jako proces pod kontrolą systemu operacyjnego**

?

Najpierw spróbuj odpowiedzieć bez zaglądania do slajdów.

Raspberry Pi 5

sprzęt + Linux + procesy + GPIO

Następny krok

Na laboratorium wykorzystamy te warstwy w praktyce: terminal, GPIO i sterowanie sprzętem z Raspberry Pi.